

## NDDL Reference

1. [NDDL CheatSheet](#)
  1. [Variables](#)
  2. [Constraints](#)
  3. [Classes](#)
  4. [Guards](#)
  5. [Iteration](#)
  6. [Resources](#)
  7. [Initial State](#)
2. [Common Pitfalls](#)

# NDDL CheatSheet

Incomplete, minimally documented summary of common NDDL syntax. Much of this code is stolen from the more complete [NDDL Reference](#). See also the [NDDL grammar](#).

## Variables

```
int v0;
float v1 = [-inff +inff];
bool v2;
string v3 = "Hi";

enum SPEED {SLOW, MEDIUM, FAST};
SPEED v4;

typedef int [1 10] INT_TEN;
INT_TEN v5;
```

Beware of using strings (see our [warning](#)).

## Constraints

For a complete list of available constraints in NDDL, see the [Constraint Library Reference](#).

Small sample of constraints:

```
eq(a, b);
leq(b, c);
addEq(b, c, d); // b + c == d
eq(b, [4 8]);
eq(a, 5);

testEQ(true, a, 5); // Asserts that a == 5
temporalDistance(t0.end, [20 100], t1.start);
```

Temporal constraints between predicates/tokens:

```
before(predA, predB);
predA.before(predB);
```

```
predA before predB;
```

Same idea with: after, meets, met\_by, equal>equals, contains, contained\_by, paralleled\_by, parallels, starts, ends, ends\_after, ends\_before, ends\_after\_start, starts\_before\_end, starts\_during, contains\_start, ends\_during, contains\_end, starts\_after, starts\_before.

## Classes

Class declaration:

```
class Navigator extends Timeline {  
    Foo f;  
    int x;  
  
    Navigator(int initVal) {  
        x = initVal;  
        f = new Foo();  
    }  
  
    predicate At{  
        Location location;  
        int x;  
        neq(x, 3);  
    }  
  
    predicate Going{  
        Location from;  
        Location to;  
    }  
  
    Navigator::At{  
        if(b == true){  
            eq(x,1);  
        }  
    }  
}
```

Rules (ie constraints on objects, their predicates and their variables):

```
Navigator::At{  
    meets(object.Going successor);  
    eq(successor.from, location);  
  
    any(Foo.p t1); // Unconstrained slave t1  
    any(Foo.p t2); // Unconstrained slave t2  
    t2 meets t1; // An Allen relation between 2 slaves  
    this meets t2; // An Allen relation between the master and the slave
```

Inheritance:

```
class Foo {  
    int arg1;  
  
    Foo() {  
        arg1 = [0 10];  
    }  
}
```

```

// Declare a subclass
class Bar extends Foo {
    string arg4;

    Bar() {
        // Must explicitly invokes superclass default constructor!
        super();
        arg1 = [5 10];
        arg4 = "empty string";
    }
}

```

## Guards

```

Bool b;
if(b == true){
    <rule statements>
}
if(b == false){
    <rule statements>
}

```

## Iteration

```

Submarine::submerge {
    Hatches hatches; // Local variable populated with the set of all hatches
    foreach(hatch in hatches) {
        contained_by(hatch.closed);
    }

    Hatches hatches filterOnly; // filterOnly keyword if it's ok if no hatches exist
    foreach(hatch in hatches) {
        contained_by(hatch.closed);
    }
}

```

## Resources

See [PLASMA/trunk/src/PLASMA/Resource/component/NDDL/Resources.nddl](#) for the complete NDDL specification of the resource classes.

```

#include "Resources.nddl"

class Battery extends Resource {
    Battery(float ic, float ll_min, float ll_max) {
        super(ic, ll_min, ll_max);
    }
}

class Oatmeal extends Reservoir { ... }

class PhoneLine extends Unary { ... }

class VanPool extends Reusable { ... }

```

Examples of all the predicates available for the above resource types:

```
Bulb::lightOn{
    starts(Battery.change tx);
    eq(tx.quantity, -600); // consume battery power
}

breakfast.starts(Oatmeal.consume c);
groceryTrip.meets(Oatmeal.produce p);

x.concurrent(PhoneLine.use x);

y.met_by(VanPool.uses vanUse);
```

## Initial State

Using classes:

```
new Foo();
Foo f1 = new Foo();
Foo f;

Foo f2 = f1; // Allocate a variable and restrict its domain to the object held in f1

Foo f3;
eq(f3, f1);
f3.reset();
```

Using predicates:

```
goal(foo.pred t0); // Initially active
rejectable(foo.pred t2); // Initially inactive
t2.reject(); // Now rejected
rejectable(foo.pred t3); // Initially inactive
t3.activate(); // Now Active
t3.cancel(); // Now inactive again

t0.location.specify(Rock);
t0.start.specify(0);

// Constraint t0 before t1 on fool
fool.constrain(t0, t1);

// Now free it to illustrate the reversal.
fool.free(t0, t1);

any(Going successor);
concurrent(this.end, successor.start);
```

Other stuff:

```
#include "MyOtherFile.nddl"

close(); // No new objects allowed
```

```
Foo.close(); // No new Foo objects allowed
```

## Common Pitfalls

- Using strings (see our [warning](#)).
- Predicates that can end up outside the horizon. If there are no constraints forcing a predicate to occur between 0 and the horizon, the planner allows them to be outside the horizon and then doesn't have to satisfy their constraints, etc. It is common therefore, to have something like `x.start.specify(0);` or `leq(0, x.start);` in predicated definitions or in the initial state.